PV112 Programování grafických aplikací

Jaro 2017

Výukový materiál

---

1. přednáška: Úvod

Co je OpenGL

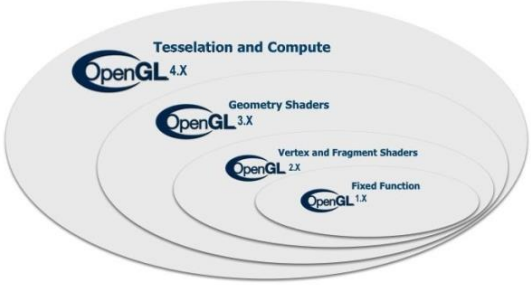
Knihovna OpenGL (Open Graphics Library) byla navržena firmou SGI (Silicon Graphics Inc.) jako aplikační programové rozhraní (API) k akcelerovaným grafickým kartám i celým grafickým subsystémům. Hlavním cílem byla co největší nezávislost na hardware (pokud možno úplná). Návrh byl vytvořen s ohledem na použití různých grafických akcelerátorů a dokonce aby bylo OpenGL možno použít i v případě, že na určité platformě žádný grafický akcelerátor není nainstalován. V takovém případě se použije softwarová simulace.

Jediným požadavkem na hardware je přítomnost framebufferu, kam se ukládají výsledná data při vykreslování pomocí OpenGL a vytváří se zde rastrová podoba výsledné scény.

Knihovna OpenGL byla vytvořena tak, aby byla nezávislá na použitém operačním systému, grafických ovladačích a správcích oken. Z toho důvodu také neobsahuje žádné funkce pro práci s okny (otevírání, zavírání, změna velikosti), pro vytvoření grafického uživatelského rozhraní (GUI) ani pro zpracování událostí. Tyto funkce lze získat pomocí volání přímo příslušného správce oken nebo je možné využít některou z nadstaveb, například v C/C++ knihovnu GLUT (OpenGL Utility Toolkit).

Na některých platformách je možné aplikaci rozdělit na dvě relativně samostatné části – serverovou a klientskou část. Při samotném vykreslování se pak jednotlivé OpenGL příkazy přenášejí přes síťové rozhraní.

Verze OpenGL



První verze OpenGL se objevila v roce 1992 a autory byli Mark Segal a Kurt Alexey. Od té doby se postupně objevovaly nové verze specifikace, které vždy určily základní vlastnosti podporovaných grafických karet pro daný release.

Pro správné pochopení základů OpenGL budeme vycházet ze specifikací nejstarších verzí. Prvním zásadním zlomem ve vývoji se stal rok 2004, kdy byly uvedeny vertex a fragment shadery v podobě tzv. GLSL (OpenGl Shading Language). Jejich programování je obsahem jiné přednášky.

V roce 2008 přišla verze 3.0 a s ní takzvaný deprecation mechanismus, který měl za cíl zjednodušit budoucí revize OpenGL API. To znamená, že některé funkce byly označeny jako deprecated. Důvodem byla především jejich zastaralost a vzhledem k tomu, že novější verze poskytly rychlejší a kvalitnější řešení, nebylo důvodem udržovat starší řešení. Mezi deprecated funkce patřily mimo jiné i například první verze GLSL a zároveň zakázání používání zpracování vertexů a fragmentů bez použití shaderů.

Verze 3.1 pak zcela odstranila funkce, které byly v 3.0 označeny jako deprecated. Aplikace, které je i dále vyžadovaly, musely využít rozšíření ARB_compatibility, kterou většina OpenGL implementací poskytuje, přestože je doporučováno se jejím funkcím vyhnout a nahradit je novějším řešením.

Odstranění zastaralé funkcionality ale díky přítomnosti ARB_compatibility rozšíření ve většině implementací problém nevyřešilo. Proto verze 3.2 přišla s konceptem tzv. profilů. Profil OpenGL definuje určitou část funkcionality OpenGL a pro danou implementaci se rozhoduje, zda bude daný profil podporovat. V současnosti existují dva profily:

- **Compatibility profil**, který implementuje kompletní OpenGL včetně deprecation modelu
- **Core profil**, který nepodporuje většinu věcí, které jsou označeny jako zastaralé.

Všechny implementace OpenGL od verze 3.2 výše pak musí poskytovat alespoň core profil.

Verze 3.3 byla vypuštěna společně s novou verzí OpenGL – 4.0. Verze 3.3 cílila na hardware, který podporoval knihovnu Direct3D 10. Verze 4.0 se soustředila na hardware podporující Direct3D 11.

Poslední verze 4.3 již umožňuje použít paralelismu GPU pro obecné výpočty. Nejnovější verzí je 4.5 z roku 2014.

16. února 2016 byl vydán konsorciem Khronos Vulkan, což je nová generace grafického a výpočetního API. Je multiplatformní a dosahuje vysoké efektivity, ale za cenu více low-levelu přístupu. Je vhodný pro PD, herní konzole, mobilní zařízení a embedded platformy. Více informací je k dispozici na stránce https://www.khronos.org/vulkan/.

Kdo vytváří specifikaci OpenGL?

OpenGL Architecture Review Board (ARB) je nezávislé konsorcium, které vzniklo v roce 1992 s cílem řídit budoucí vývoj OpenGL, navrhovat a schvalovat změny ve specifikaci a plánovat nové releases. Od roku 2006 se o tyto úkoly stará Khronos Group (www.khronos.org), která se stala novým „domovem“ ARB. K tomuto přesunu došlo na základě odhlasování členskými institucemi, kterými jsou 3Dlabs, Apple, ATI, Dell, IBM, Intel, NVIDIA, SGI a Sun Microsystems. Jedním z původních členů byl i Microsoft, který však v roce 2003 ARB opustil.

Nové OpenGL knihovny mohou vytvářet libovolní poskytovatelé hardware, musí však projít tzv. conformance testy. OpenGL Conformance Tests je sada testů, které OpenGL ARB využívá k ověření, že daná implementace vyhovuje specifikaci OpenGL a poté, pokud autor zaplatí licenční poplatek, může se tato knihovna nazývat OpenGL knihovnou. Conformance testy odhalí velké množství bugů, nicméně nemohou zaručit, že kód, který jimi projde, je bezchybný.

První pohled na OpenGL

OpenGL je považováno za procedurální interface, což znamená, že obsahuje přímo výkonné instrukce na úrovni programovacího jazyka. Na druhé straně stojí deskriptivní interface, jehož příkladem je grafický formát VRML. Myšlenka deskriptivního rozhraní spočívá v popisu obsažených informací, neobsahuje tedy konkrétní instrukce. Deskriptivní interface lze vybudovat na základě procedurálního, ale ne naopak.

Dále je třeba si uvědomit, že OpenGL není tzv. „pixelově exaktní“. To znamená, že OpenGL nezaručuje, že při spuštění identického programu, který používajícího OpenGL knihovnu na různých platformách nebo různých grafických akcelerátorech dostaneme přesně stejný výsledek. Při porovnání výsledných obrázku po pixelech můžeme objevit mírné rozdíly v barvách. To může být způsobeno například odlišnou přesností reprezentace čísel na grafické kartě (int, float) různými algoritmy pro interpolaci barvy, normály, texturových souřadnic nebo jinou bitovou hloubkou Z-bufferu. Celková geometrie a barevnost scény by však měla odpovídat.

Co OpenGL umí?

OpenGL (alespoň jeho starší verze) poskytuje následující funkce:

- vykreslování trojúhelníků, bodů
- texturování (texturing)

- stínování (shading) – zde myšleno jako výběr osvětlovacího modelu. Jinak jsou již používány shadery.
- výpočet viditelnosti
- alfa míchání (alpha blending)
- akumulační buffer (accumulation buffer)
- buffer šablony (stencil buffer)

Následující funkce jsou podporovány, je třeba však vytvořit vlastní implementaci:
- transformace (transformations)
- mlha (fog)
- osvětlení (lighting)
- stíny (shadows)
- odrazy
- voxely

Co OpenGL neumí?

Naopak OpenGL nepodporuje následující funkce:

- práci s okny
- NURBS (lze je však použít za využití Compute shaderů nebo aproximací přes Tesselation shadery)
- reprezentaci scény (scene representation)

Knihovny OpenGL

Nejstarší verze OpenGL obsahovaly knihovu GLU (OpenGL Utility Library), která poskytuje nadstavbové velmi užitečné funkce, jako například mipmapping, teselace a generování jednoduchých tvarů. GLU specifikace byla naposledy updatována v roce 1998 a poslední verze závisí zejména na deprecated funkcích z verze OpenGL 3.1 v roce 2009.

S ohledem na to, že manuální vytvoření OpenGL kontextu je poměrně složitý proces a navíc se pro jednotlivé operační systémy tento proces liší, stalo se automatické vytváření OpenGL kontextu samozřejmostí několika vývojářských knihoven (pro gaming, GUI atd.), jako například SDL, Allegro, SFML, FLTK, Qt.

Některé knihovny byly vyvinuty pouze za účelem vytvoření OpenGL okna pro zobrazení. První takovou knihovnou byl GLUT, jeho novější alternativou je knihovna GLFW.

Pro další usnadnění a urychlení práce s OpenGL se objevila sada knihoven pro automatické natahování všech dostupných rozšíření a funkcí, jako například GLEE, GLEW. Automatické natahování rozšíření podporuje i většina nadstavbových knihoven pro další programovací jazyky, jako například JOGL (Java) nebo PyOpenGL (Python).

OpenGL a práce s okny

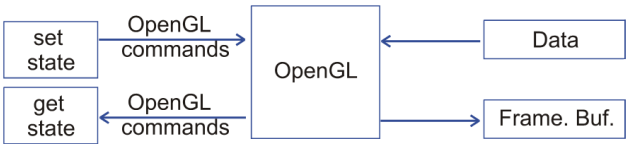
Jak již bylo řečeno, samotné OpenGL nepodporuje práci s okny z důvodu co největší hardwarové a platformní nezávislosti. V různých programovacích jazycích se proto používá řada různých nástrojů pro práci s okny. Ve spojení s jazykem C/C++ jsou to zejména knihovny:

- GLUT (OpenGL Utility Toolkit) – poskytuje funkcionalitu nezávislou na platformě, jako například manipulaci s okny, jednoduché menu, práci s kurzorem myši apod.
- SDL (Simple DirectMedia Layer) – multplatformní knihovna navržena pro nízkoúrovňový přístup ke zvuku, klávesnici, myši, joysticku, 3D hardware. Je psána v C, ale nativně pracuje s C++. Zároveň byly vytvořeny bindings na další jazyky včetně C# a Javy.
- JOGL (Java Binding for the OpenGL API) + Swing – JOGL je knihovna pro hardwarovou podporu 3D grafiky pro aplikace psány v Javě. JOGL poskytuje plnou funkcionalitu OpenGL od verze 1.3.

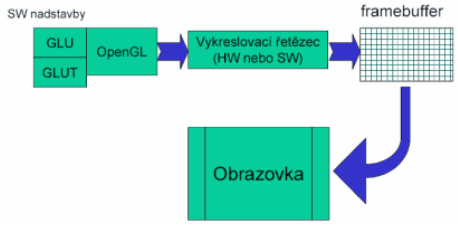
Samozřejmě existuje celá řada dalších možností, jak pracovat s okny v OpenGL, jako například X-Window system používající GL widget class (glx.h) a další. Ale těmi se v rámci tohoto kurzu zabývat nebudeme.

OpenGL jako stavový automat

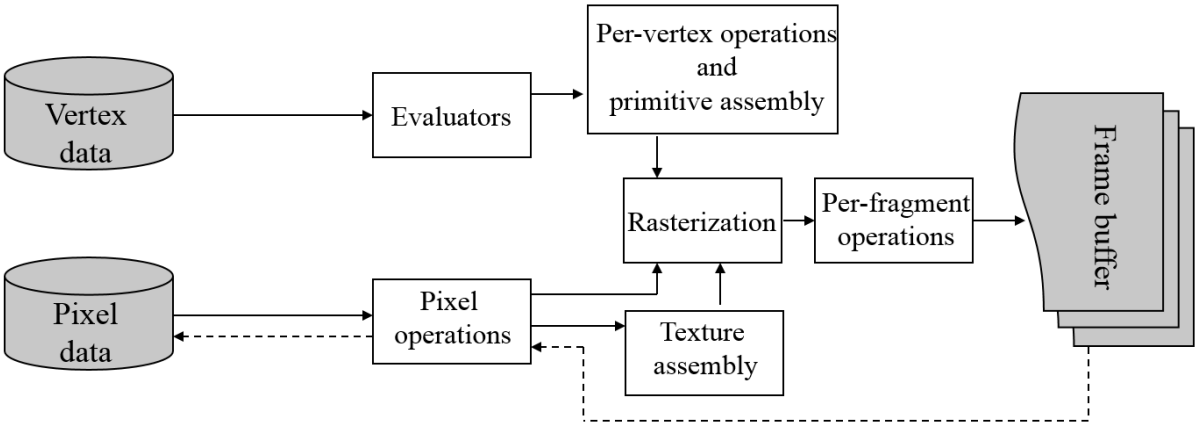
Z programátorského hlediska se OpenGL chová jako stavový automat. To znamená, že během zadávání jednotlivých příkazů pro vykreslování je možné průběžně měnit vlastnosti vykreslovacích primitiv (například barva nebo průhlednost) nebo celé scény (transformace, volby způsobu vykreslování apod). Toto nastavení je zachováno, dokud jej opět nezměníme. Výhodou tohoto přístupu je především menší počet parametrů funkcí pro vykreslování a skutečnost, že jedním příkazem můžeme globálně změnit způsob vykreslení celé scény (např. nastavení použití drátového – wireframe modelu, zobrazení pomocí vyplněných polygonů – filled model).



Jak už bylo zmíněno, vykreslování scény se provádí procedurálně – voláním funkcí OpenGL vzniká výsledný rastrový obrázek, který je uložen do tzv. **framebufferu**, kde je každému pixelu přiřazena barva, hloubka, alfa složka (průhlednost) a další atributy. Z framebufferu je možné získat pouze barevnou informaci a tu je možné následně zobrazit na obrazovce.



Blokový diagram OpenGL



Blokový diagram na obrázku, známý též jako **OpenGL Rendering Pipeline**, představuje abstraktní vysokoúrovňový pohled na to, jakým způsobem OpenGL zpracovává data. Diagram se skládá z jednotlivých modulů, které pracují relativně nezávisle a mezi moduly se nepředávají žádné řídicí informace. Vše je řízeno pouze daty (proto je tato architektura označována jako „data-flow“), která do pipeline postupně vstupují. Z toho také vyplývá rozdíl například vůči klasickému procesoru (řízen programem) a rovněž velká rychlost vykreslování pomocí grafických akcelerátorů.

Každý modul provádí paralelně jednu instrukci (princip pipeline) – velmi efektivní a rychlé, zvláště pro velké datové množiny.

Vstupní data mohou být dvojího druhu – vertex data reprezentující geometrii a pixel data. Geometrická data dále prochází do fáze, která je označena jako **Evaluators**. Představuje

efektivní prostředek pro aproximaci geometrie křivek a povrchů, ke které dochází vyhodnocením polynomiálních příkazů vstupních hodnot.

V další fázi, **Per-vertex operations and primitive assembly**, OpenGL zpracovává geometrická primitiva – body, čárové segmenty a polygony, které jsou popsány vrcholy (vertices). V části „**Per-vertex operations**“ jsou souřadnice každého vrcholu a jeho normály transformovány pomocí matice (z lokálních souřadnic objektu do souřadnic pozorovatele (eye coordinates)). Pokud je povoleno osvětlení, je v této fázi spočteno pro každý vrchol. Výpočet osvětlení rovněž aktualizuje barvu daného vrcholu.

V části „**Primitive assembly**“ dochází k transformaci grafických primitiv (bod, čára, polygon) pomocí projekční matice a dále mohou být vertexy ořezány některou předem zadanou ořezávací rovinou. Pokud je vertex umístěn za ořezávací rovinou, je odstraněn z dalšího zpracování. Při ořezávání čar nebo polygonů mohou být naopak další vertexy přidány, aby se vykreslil zbytek viditelné části primitiva.

Tímto jsou data připravena na další fázi.

Pro pixelová vstupní data je opět možné využít display listů pro odložené renderování. Poté následuje fáze zvaná **Pixel operations**, kde je na vstupních pixelech proveden příslušný scaling, bias (posunutí), mapování atd.

Takto zpracované pixely jsou buď uloženy v tzv. **Texture assembly** (**Texture memory**) jako rastr textury nebo jsou rovnou poslány do rasterizační jednotky. Texture memory uchovává texturové obrázky, které pak mohou být aplikovány na geometrické objekty.

Data upravená oběma popsanými postupy vstupují do procesu **Rasterization**. Rasterizace je proces převodu geometrických a pixelových dat na takzvané fragmenty. Fragmenty jsou dvoudimenzionální obdélníkové oblasti obsahující barvu, hloubku, šířku čáry, velikost bodu a výpočet antialiasingu. Každý fragment odpovídá pixelu ve framebufferu.

Každý takto vytvořený fragment je zpracován ve fázi **Per-fragment operations**, kde je převeden na pixel ve framebufferu. První část této fáze je generování tzv. texelů (texture elements). Texture element je generován z texture memory a je aplikován na každý fragment. Dále se vypočítává mlha. Následuje sada testů na fragmentech: Scissor test => Alpha test => Stencil test => Depth test. Nakonec je proveden blending (míchání), dithering, logické operace, aplikace bitové masky a výsledné pixely jsou uloženy do framebufferu.

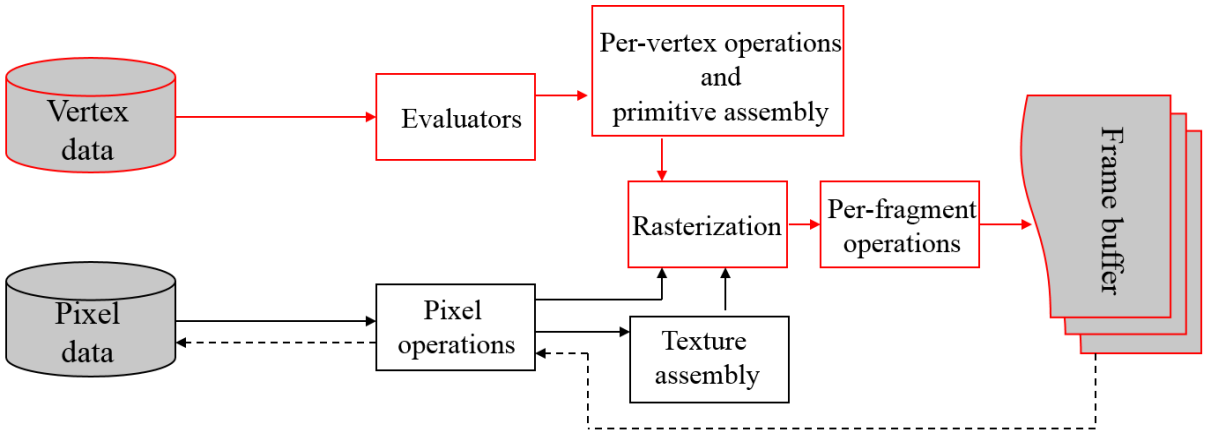
Framebuffer se skládá z několika samostatných bufferů – nejdůležitější jsou color buffer, Z-buffer (pamět hloubky), stencil buffer (pamět šablony), accumulation buffer (akumulační buffer). Jednotlivé buffery budou detailně popsány později.

Vykreslování trojúhelníka

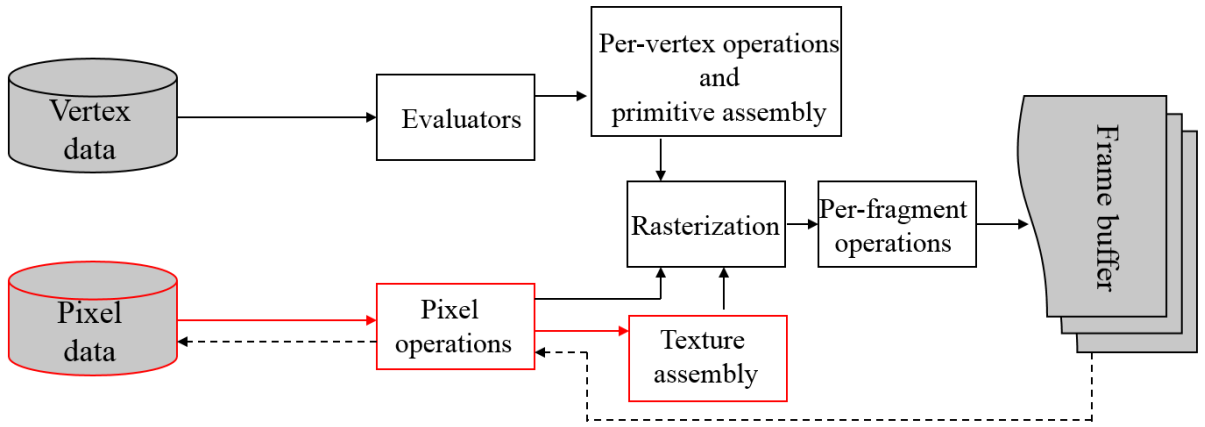
Jako příklad uvedeme vykreslování trojúhelníka, nejdříve bez textury a následně s texturou.

Trojúhelník bez textury je definován pomocí vrcholů, proto jsou vstupní data typu vertex. Data prochází přes evaluátor do modulu Per-vertex operations and Primitive assembly. Následující krok (již společný pro vertex i pixel vstupní data) je rasterizace. Poté jsou aplikovány per-fragment operace a pixely jsou uloženy do framebufferu.

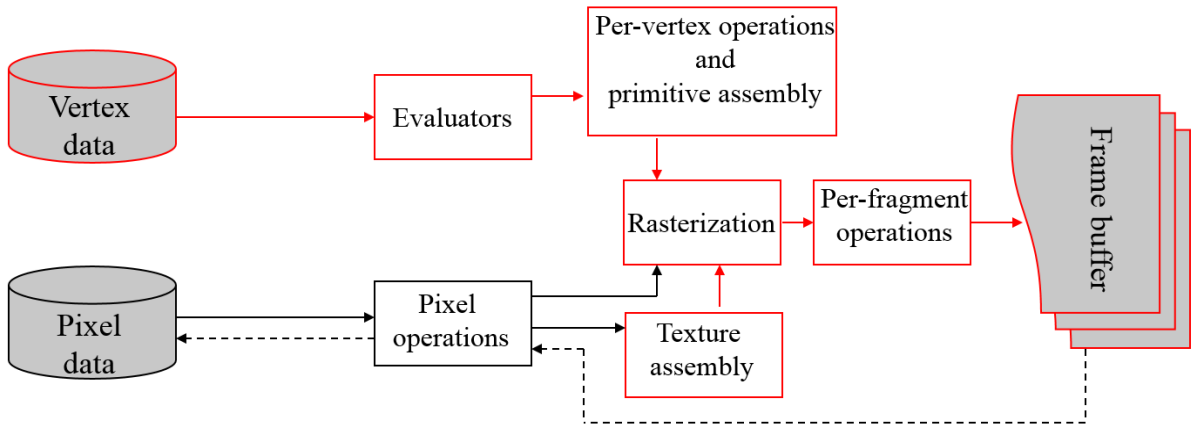
Z celého procesu je nejnáročnější proces rasterizace.



Pokud chceme vykreslit **trojúhelník opatřený texturou**, je proces průchodu grafickou pipeline OpenGL jiný. Nejdříve je v první fázi připravena textura definovaná pixelovými daty a je uložena do paměti Texture assembly. Tím je připravena pro další použití v následujícím kroku.

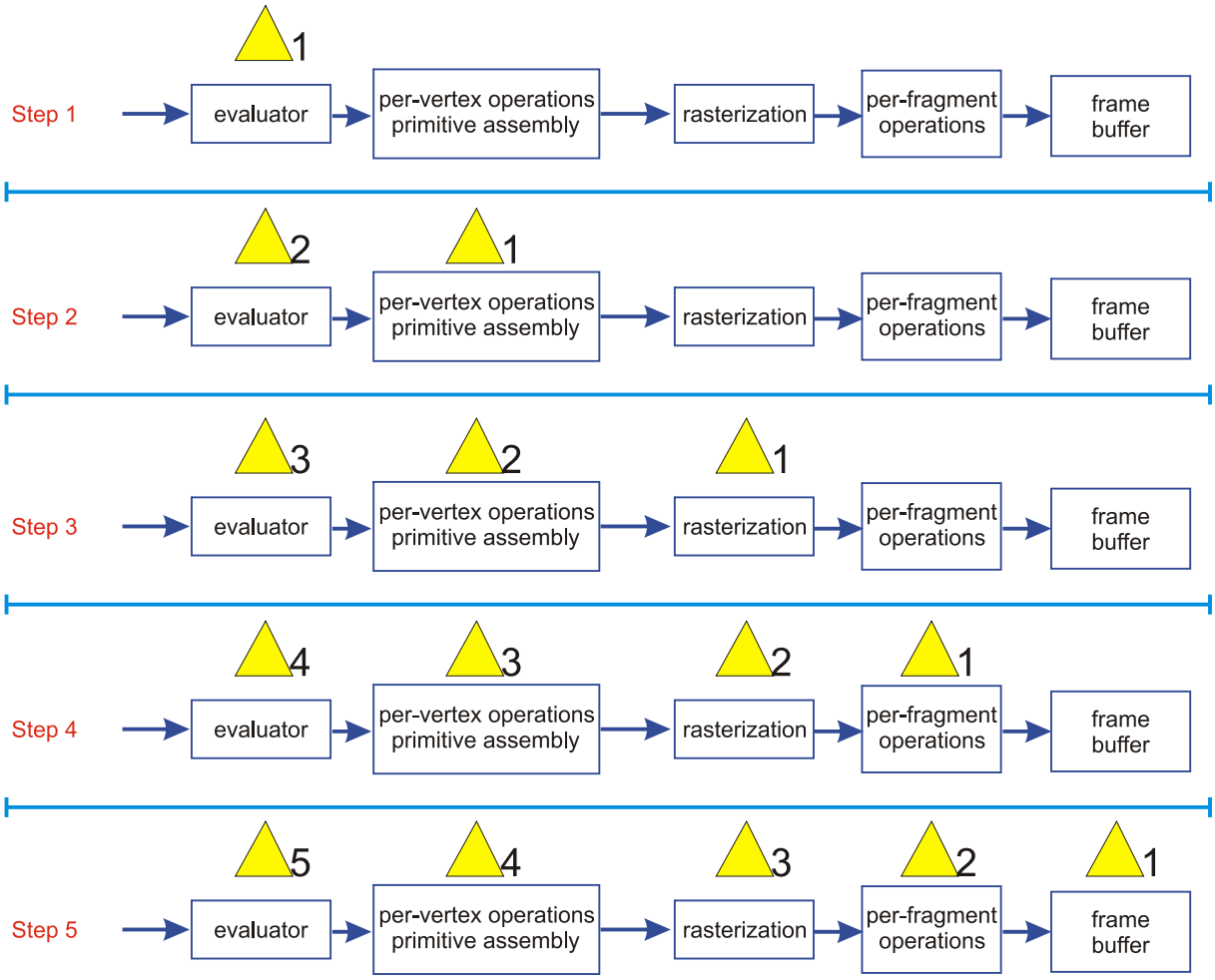


Ve druhé fázi jsou opět stejným způsobem jako v prvním případě zpracována geometrická data definující požadovaný trojúhelník, ale v procesu rasterizace jsou do zpracování zahrnuty informace o textuře uložené v Texture assembly.



Pipelining – vykreslování množiny trojúhelníků

Pro demonstraci principu pipeliningu využijeme následující obrázek.



V prvním kroku vstupuje trojúhelník 1 do modulu Evaluator. Až je zpracován, posune se trojúhelník 1 do druhého modulu, Per-vertex operations and primitive assembly a zároveň

do prvního modulu vstupuje nový trojúhelník 2. Tak to pokračuje dále, až v pátém kroku jsou zaměstnány všechny moduly a je souběžně zpracováváno 5 trojúhelníků.

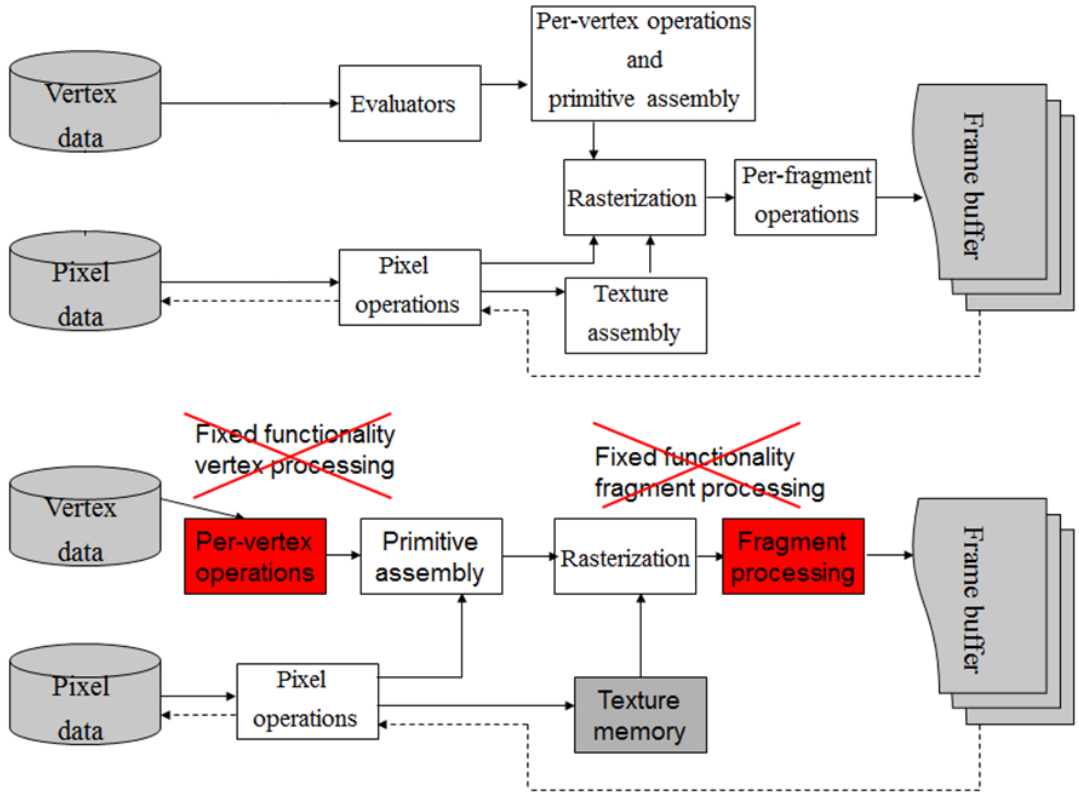
GLSL (Graphics Library Shading Language)

Jedná se o vysokoúrovňový jazyk pro psaní shaderů. Tímto způsobem nahrazujeme některé fixní části renderovací pipeline programovatelnými částmi, které jsou psány právě pomocí shaderů. GLSL program je pak program složený ze sady shaderů, které jsou zkompilovány a „slinkovány“. Syntaxe GLSL jazyka se podobá jazyku C, je obohacena o datové typy vektor a matice a využívá některých výhod jazyka C++.

V současné době již lze použít několik různých shaderů nahrazujících jednotlivé části fixní pipeline. Jedná se o:

- **Vertex shader**
- **Fragment shader**
- Tesselation shader
- Geometry shader
- Compute shader

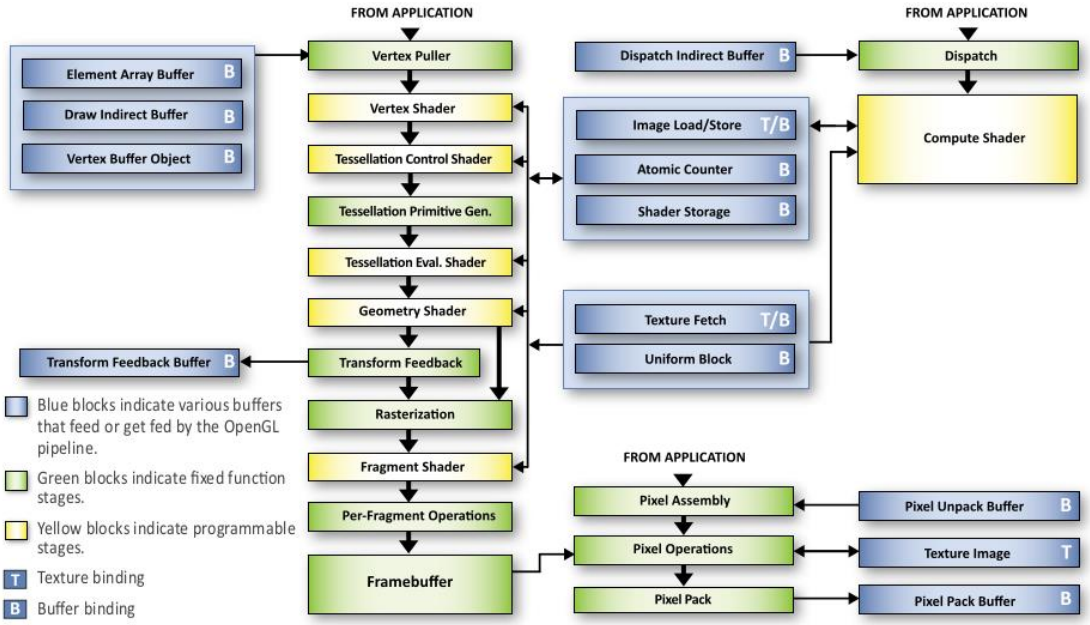
V rámci tohoto předmětu se budeme zabývat pouze vertex a fragment shadery.



Shader program

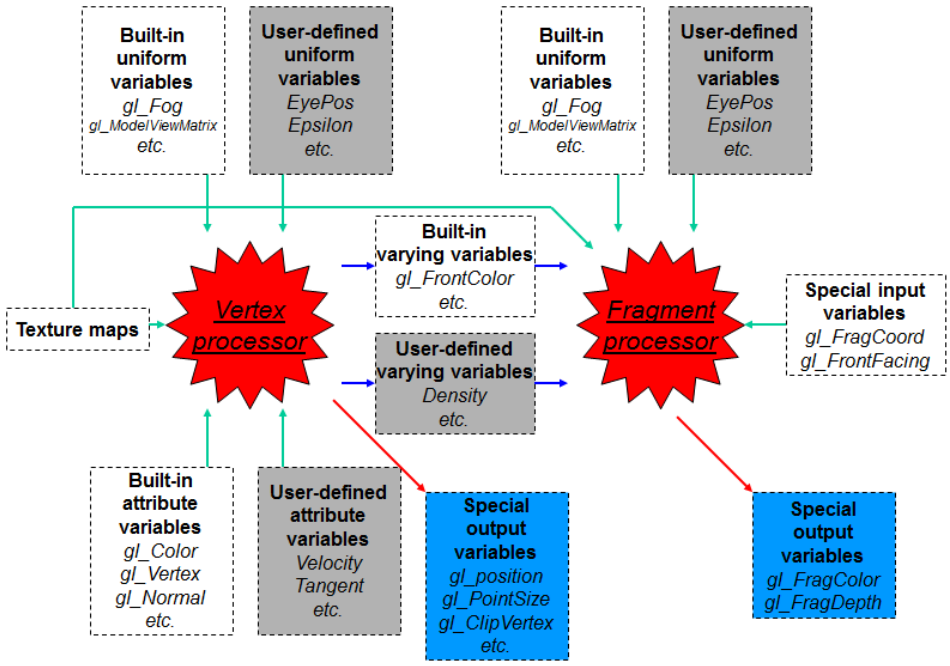
Jde o malý program řídící určitou část grafické pipeline. Shader je kód, který je prováděn pro každý vrchol (vertex shader) nebo fragment = pixel (fragment shader) přímo na GPU. Vertex a fragment shadery mezi sebou navzájem komunikují. Vertex shader se stará o transformace vrcholů, barvu těchto vrcholů, osvětlení na úrovni vrcholů a další funkce. Fragment shader se pak zabývá texturováním či barvou a osvětlením na úrovni jednotlivých pixelů.

Moderní grafická pipeline sestávající z různých typů shaderů pak vypadá následovně:



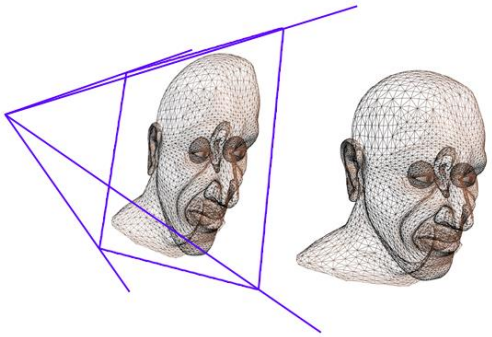
GLSL program poté specifikuje, jak má OpenGL vykreslovat geometrii. GLSL program sestává z kolekce shaderů, přičemž musí obsahovat alespoň jeden vertex shader a alespoň jeden fragment shader. Je důležité vědět, že na GPU nemůže běžet více takovýchto programu současně. Před samotným vykreslením geometrie je tedy nutné specifikovat, který program se má provádět.

Následující obrázek ilustruje jednotlivé typy vstupů a výstupů, které lze ve vertex a fragment shaderech použít. Detaily budou následovat dále v textu.



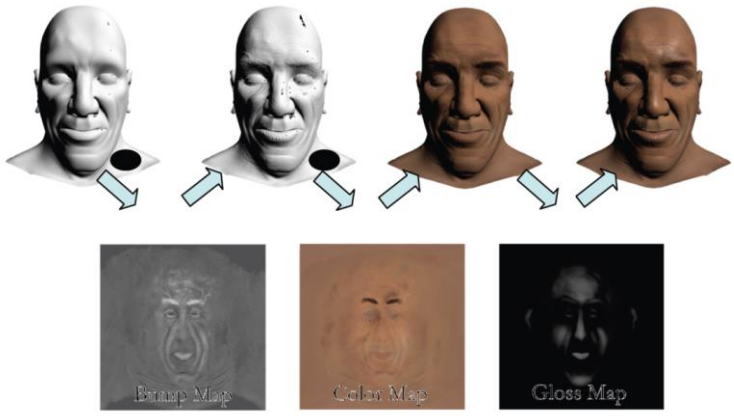
Vertex shader

Vertex shader transformuje vrcholy z prostoru objektu do prostoru obrazovky. Transformuje vrcholy, normály, či texturové souřadnice. Ve vertex shaderech může docházet ke generování texturových souřadnic, počítání osvětlení jednotlivých vrcholů, nastavení hodnot pro interpolaci, která je poté provedena ve fragment shaderu, atd. Vertex shader naopak nemá žádné informace o organizaci scény (o viewportu, ořezání okna, …).



Fragment shader

Fragment shader, někdy označován též jako pixel shader, počítá a nastavuje barvu fragmentu (neboli pixelu). Na vstupu bere data z vertex shaderu. Umí získat hodnotu barvy z textur, probíhá v něm výpočet mlhy, využívá interpolovaná data z vertex shaderu. Fragment shader naopak neumí např. změnit souřadnice fragmentů.

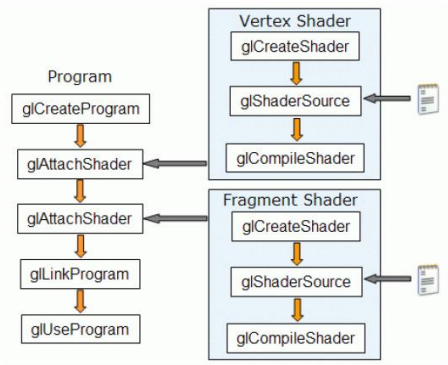


Jak vytvořit GLSL program

Vytvoření probíhá v sedmi základních krocích:

1. Vytvoření shader objektů.
2. Načtení zdrojového kódu ze souboru a naplnění vytvořených shader objektů tímto kódem.
3. Kompilace shaderu.
4. Vytvoření objektu programu.
5. Připojení (attach) shaderů k programu.
6. Slinkování programu.
7. Oznámení OpenGL, aby používalo tento program.

Shadery se vytváří a nastavují následujícím způsobem, který je ilustrován na obrázku:

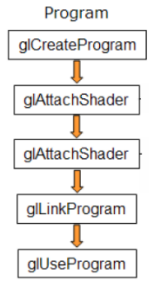


V prvním kroku je vytvořen objekt, který funguje jako kontejner shaderu. Funkce glCreateShader sloužící tomuto účelu vrací ID (jméno) tohoto kontejneru. Lze vytvořit libovolné množství shaderů, které lze připojit k programu, ale pro jeden program může být pouze jedna main funkce pro sadu vertex shaderů a jedna main funkce pro sadu fragment shaderů.

Dalším krokem je přidat shaderu zdrojový kód (pomocí glShaderSource). Zdrojový kód shaderu je v podstatě pole řetězců, avšak je možné použít pouze ukazatel na jediný řetězec.

Nakonec je nutné shader zkompilovat (pomocí glCompileShader).

Vytvoření shader programu využívajícího takto připravené shadery poté sestává z následujících kroků:



V prvním kroku je vytvořen objekt, který vystupuje jako kontejner programu. Funkce glCreateProgram plnící tuto funkci vrací ID tohoto kontejneru. Podobně jako u shaderu zde platí, že můžeme vytvořit libovolné množství programů. Při renderování je pak možné mezi těmito programy přepínat. Například je takto možné renderovat scénu obsahující konvičku za použití refrakčního a reflexního shaderu, zatímco na pozadí pomocí fixní funkcionality OpenGL je vykreslena cubemap.

Další krok zahrnuje přiřazení shaderů, které byly vytvořeny před chvílí, k právě vytvořenému programu. V této fázi není nutné, aby přiřazené shadery byly již zkompilovány – dokonce ani nemusí mít zatím přiřazeny zdrojový kód. Jediné, co je potřeba k přiřazení shaderu danému programu, je odpovídající kontejner shaderu.

Pokud máme vytvořen pár vertex a fragment shaderu, chceme je oba přiřadit danému programu. Můžeme mít vytvořenu řadu vertex a fragment shaderů, které přiřadíme danému programu (podobně jako v C může mít program mnoho modulů). Ale pro každý typ shaderu můžeme mít pouze jeden shader obsahující main funkci (jako v C).

Jeden shader je možné přiřadit několika různým programům – například když chceme stejný vertex shader použít ve více aplikacích.

Posledním krokem je přilinkování programu. V tomto kroku je již nutné, aby byly použité shadery zkompilovány. Po provedení této operace je možné modifikovat zdrojový kód shaderu a znovu shader překompilovat – tento krok neovlivní program.

Po přilinkování programu již následuje jeho samotné použití. Každému programu je přiřazen ukazatel. Můžeme takto použít libovolné množství přilinkovaných a připravených programů – jejich počet je omezen použitým hardware.

Podobně jako byl programu shader přiřazen, je možné jej odpojit. K tomu slouží funkce glDetachShader. Pokud chceme nějaký shader smazat, musí být odpojen – tato funkce je tedy důležitá.

Pokud chceme daný shader nebo program smazat, použijeme odpovídající z funkcí glDeleteShader nebo glDeleteProgram.

V případě, že je daný shader stále přiřazen k nějakému (nebo i více) programu, není shader smazán, pouze označen pro smazání. Opravdu smazán bude shader až poté, co nebude přiřazen k žádnému programu (byl tedy uvolněn ze všech programů, které ho používaly).

Následující zdrojový kód obsahuje všechny kroky, které jsme si popsali. Proměnné *p*, *f* a *v* jsou deklarovány jako globální jako GLuint.

```
void setShaders()
{
  char *vs, *fs;

  // Setup
  v = glCreateShader(GL_VERTEX_SHADER);
  f = glCreateShader(GL_FRAGMENT_SHADER);

  vs = textFileRead("simple.vert");
  fs = textFileRead("simple.frag");

  const char * vv = vs;
  const char * ff = fs;

  glShaderSource(v, 1, &vv, NULL);
  glShaderSource(f, 1, &ff, NULL);

  free(vs);
  free(fs);

  glCompileShader(v);
  glCompileShader(f);

  p = glCreateProgram();

  glAttachShader(p, v);
  glAttachShader(p, f);

  glLinkProgram(p);
  glUseProgram(p);

  ...

  // Clean up
  glDetachShader(p, v);
  glDetachShader(p, f);

  glDeleteShader(v);
  glDeleteShader(f);

  glUseProgram(0);
  glDeleteProgram(p);
}
```

Nástroje pro psaní shaderů

Protože shadery jsou v podstatě pouze textové řetězce, lze použít libovolný textový editor. Mezi nástroje používané často pro psaní shaderů patří:

- RenderMonkey (http://developer.amd.com/tools-and-sdks/archive/legacy-cpu-gpu-tools/rendermonkey-toolsuite/)
- FX composer (https://developer.nvidia.com/fx-composer)
- OpenGL Shader Designer (http://www.opengl.org/sdk/tools/ShaderDesigner/)
- NVIDIA Nsight
    - Pouze pro registrované vývojáře
- AMD CodeXL
- gDEBugger (http://www.gremedy.com/)
- Visual Studio
    - Zvýraznění syntaxe, IntelliSense
    - Používá se v PV227 (navazující předmět)

GLSL datové typy

Mezi základní datové typy patří:

- float, int
- vec2, vec3, vec4: vektory
- mat2, mat3, mat4: matice
- sampler1D, sampler2D, sample3D, samplerCube, …: textury

vec2 = vektor ve 2D obsahující float hodnoty

```
vec2 a;
a.x = 0.0;
a.y = 1.0; // a = (0,1)

vec2 b;
b.s = 10.0;
b.t = 12.5; // b = (10,12.5)

vec2 c;
c[0] = 9.0;
c[1] = 8.0; // c= (9,8)
```

vec3 = vektor ve 3D, jako x, y, z se označují souřadnice pozice, r, g, b souřadnice barvy a s, t, p texturové souřadnice

```
vec3 a;
a.x = 10.0; a.y = 20.0; a.z = 30.0;  // a = (10,  20,  30)
a.r = 0.1;  a.g = 0.2;  a.b = 0.3;   // a = (0.1, 0.2, 0.3)
a.s = 1.0,  a.t = 2.0;  a.p = 3.0;   // a = (1, 2, 3)

vec3 b = vec3(4.0, 5.0, 6.0);

vec3 c = a + b; // c = (5, 7, 9)
vec3 d = a - b; // d = (-3, -3, -3)
vec3 e = a * b; // e = (4, 10, 18)
vec3 f = a * 3; // e = (3, 6, 9)
float g = dot(a,b); // g = 32
vec3 h = cross(a,b); // h = (-5,6,-3)
float i = length(a); // i = 3.742
```

vec4 = vektor ve 4D, navíc do souřadnic vstupují ještě složky w, resp. a, resp. q

```
vec4 a;
a.x = 10.0; a.y = 20.0; a.z = 30.0; a.w = 40.0;
// a = (10, 20, 30, 40)
a.r = 0.1;  a.g = 0.2;  a.b = 0.3;  a.a = 0.4;
// a = (0.1, 0.2, 0.3, 0.4)
a.s = 1.0;  a.t = 2.0;  a.p = 3.0;  a.q = 4.0;
// a = (1, 2, 3, 4)

vec4 b = vec4(5, 6, 7, 8);

vec4 c = a + b; // c = (6, 8, 10, 12)
vec4 d = a - b; // d = (-4, -4, -4, -4)
vec4 e = a * b; // e = (5, 12, 21, 32)
vec4 f = a * 3; // f = (3, 6, 9, 12)
float g = length(a); // g = 5.477
```

mat2 = matice 2x2 obsahující float čísla

```
mat2 A = mat2(1.0, 2.0, 3.0, 4.0); // in column-major order

vec2 x = vec2(1.0, 0.0);
vec2 y = vec2(0.0, 1.0);

vec2 a = A * x; // a = (1,2)
vec2 b = A * y; // b = (3,4)
```

mat3 = matice 3x3

```
mat3 A = mat3(1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0);
// in column-major order

vec3 x = vec3(1.0, 0.0, 0.0);
vec3 y = vec3(0.0, 1.0, 0.0);
vec3 z = vec3(0.0, 0.0, 1.0);

vec3 a = A * x; // a = (1,2,3)
vec3 b = A * y; // b = (4,5,6)
vec3 c = A * z; // c = (6,7,8)
```

mat4 = matice 4x4, do které lze ukládat afinní transformace

```
mat4 A = mat4(1.0,  2.0,  3.0,  4.0,
              5.0,  6.0,  7.0,  8.0,
              9.0,  10.0, 11.0, 12.0
              13.0, 14.0, 15.0, 16.0); // in column-major order

vec4 x = vec4(1.0, 0.0, 0.0, 0.0);
vec4 y = vec4(0.0, 1.0, 0.0, 0.0);
vec4 z = vec4(0.0, 0.0, 1.0, 0.0);
vec4 w = vec4(0.0, 0.0, 0.0, 1.0);

vec4 a = A * x; // a = (1,2,3,4)
vec4 b = A * y; // b = (5,6,7,8)
vec4 c = A * z; // c = (9,10,11,12)
vec4 d = A * w; // d = (13,14,15,16)
```

Pole (array) = má fixní velikost, notace převzata z jazyka C

```
float A[4];
A[0] = 5; A[3] = 10;

vec4 B[10];
B[3] = vec4(1,2,3,4);
B[8].y = 10.0;
```

Typy vstupů a výstupů

Shadery mohou mít tři různé typy vstupů a výstupů:

- **uniforms**
- **attributes (in)**
- **varyings (out)**

Uniforms

Uniforms jsou hodnoty, které se během renderování nemění (například pozice světla či barva světla). Jsou dostupné ve vertex i fragment shaderech. Jsou read-only.

Attributes (in)

Attributes jsou dostupné pouze ve vertex shaderu a jsou to vstupní hodnoty, které mění každý vrchol (například pozice vrcholu nebo normály). Jsou rovněž read-only.

Varyings (out)

Varyings jsou používány pro transfer dat z vertex shaderu do fragment shaderu. Varyings jsou read-only ve fragment shaderu, ale jsou read a write ve vertex shaderu (ale pozor – čtení varying typu před zápisem do něj vrátí nedefinovanou hodnotu). Pokud chceme varyings používat, musíme deklarovat stejnou varying ve vertex i fragment shaderu.

Můžeme využít i řadu vestavěných typů, které jsou používány pro výstup shaderu. My budeme využívat:

gl_Position	4D vektor reprezentující finální pozici zpracovaného vrcholu. Dostupné pouze pro vertex shader.

Generické typy

Můžeme si rovněž specifikovat své vlastní vstupních, výstupních či uniforms. Příklady:

```
uniform sampler2D my_color_texture;
uniform mat4 my_texture_matrix;

in vec3 tangent;
in vec3 binormal;

out vec3 vertex_to_light_vector;
out vec3 vertex_to_eye_vector;
```

Inicializace a konstruktory

Proměnná shaderu může být inicializována v okamžiku její deklarace. Jakmile ji však změním pomocí C++/Java kódu, použije se tato nová hodnota. To neplatí pro proměnné typu **in**, **out** a **uniform**.

```
vec4 v = vec4(1.0, 0.5, 0.0, 1.0);
---
vec4 v;
v = vec4(1.0, 0.5, 0.0, 1.0);
mat2 m = (1.0, 2.0, 3.0, 4.0);
---
```

Existují konstruktory pro všechny vestavěné typy (s výjimkou samplerů) a pro **struktury**:

Struktury

Uživatelem definované typy je možné vytvářet agregací předdefinovaných typů do struktur uvozených klíčovým slovem **struct**. Příklad:

```
struct light
{
     float intensity;
     vec3 position;
} lightVar;
```

Kde *light* je jméno nového typu a *lightVar* je proměnnou typu *light*. Pro deklaraci proměnné tohoto nového typu stačí použít její jméno:

```
light lightVar2;
```

Příklad:

```
vec3 color = vec3(0.2, 0.5, 0.6);
vec4 v = vec4(20.0, 12.4, 2.1, 1.0);
struct light
{
  vec4 position;
  struct lightColor
  {
    vec3 color;
    float intensity;
  }
} light1 = light(v, lightColor(color, 0.9));
```

Typová konverze

Explicitní typová konverze se rovněž provádí pomocí konstruktorů, protože GLSL neposkytuje syntaxi pro převod typů.

Příklad:

```
float f = 2.3;
bool b = bool(f);
float f = float(3); // convert integer 3 to 3.0
float g = float(b); // convert Boolean b to floating point
vec4 v = vec4(2)    // all components of v are set to 2.0
```

Přístup ke komponentám

Ke komponentám jednotlivých proměnných je možné přistupovat pomocí tečkové notace:

```
vec2 pos;
pos.x  // is legal
pos.z // is illegal
```

Syntaxe povoluje rovněž výběr několika komponent najednou – uvedením jejich jména (označení) za tečkou:

```
vec4 v4;
v4.rgba; // is a vec4 and the same as just using v4
v4.rgb; // is a vec3
```

```
v4.b; // is a float
v4.xy; // is a vec2
v4.xgba; // is illegal - the component names do not come from the
same set
```

Swizzling

Pořadí jednotlivých komponent může být změněno nebo replikováno pomocí tzv. swizzlingu:

```
vec4 pos = vec4(1.0, 2.0, 3.0, 4.0);
vec4 swiz = pos.wzyx; // swiz = (4.0, 3.0, 2.0, 1.0)
vec4 dup = pos.xxyy;  // dup = (1.0, 1.0, 2.0, 2.0)
```

Operace po komponentách

Kromě několika málo výjimek se operátory chovají, jako by byly aplikovány na každou komponentu vektoru zvlášť.

```
vec3 u, v, w;
float f;
v = u + f; ~ v.x = u.x + f; v.y = u.y + f; v.z = u.z + f;
w = u + v; ~ w.x = u.x + v.x; w.y = u.y + v.y; w.z = u.z + v.z;
```

Logické operátory !, &&, ||, ^^ fungují pouze na výrazech, které jsou označeny jako scalar booleans a jejich výsledkem je opět scalar boolean.

Relační operátory <, >, <=, >= fungují pouze pro floating-point a integer skaláry a výsledkem je scalar boolean.

Operátory rovnosti ==, != fungují pro všechny typy s výjimkou polí. Výsledkem je scalar boolean.

Výrazy a funkce

V GLSL můžeme používat řadu výrazů, které jsou velmi podobné jazyku C:

```
if (bool expression)
      ...
else
      ...

for (initialization; bool expression; loop expression) ...

while (bool expression) ...
do
       ...

while (bool expression)
```

Skoky jsou rovněž definovány:

- **continue** – dostupné ve smyčkách, způsobí skok na další iteraci smyčky
- **break** – dostupné ve smyčkách, ukončí zpracování smyčky
- **discard** – dostupné pouze ve fragment shaderech. Způsobí ukončení shaderu pro aktuální fragment bez jeho zápisu do framebufferu.

Podobně jako v jazyce C je shader strukturován do funkcí. Každý shader musí obsahovat minimálně main funkci deklarovanou následovně:

```
void main()
```

Rovněž můžeme definovat svoje vlastní funkce. Ty mohou mít svoji návratovou hodnotu a v tom případě by měly využívat return statement pro předání výsledků. Samozřejmě tato funkce může být i typu void. Návratový typ může být libovolný z definovaných, jediným omezením je, že návratovou hodnotou nemůže být pole.

Parametry funkce mají následující dostupné kvalifikátory:

- **in** – pro vstupní parametry.
- **out** – pro výstupy funkce. Jiným způsobem předání návratové hodnoty je využití právě výše zmíněného return statement.
- **inout** – pro parametry, které jsou zároveň vstupem i výstupem dané funkce.

Pokud není specifikován žádný kvalifikátor, je defaultně brán kvalifikátor *in*.
Další poznámky:

- Funkce může být přetížena – obsahuje jiný seznam parametrů.
- Rekurzivní chování není ve specifikaci definováno.

Příklad funkce:

```
vec4 toonify(in float intensity) {
      vec4 color;
      if (intensity > 0.98)
            color = vec4(0.8,0.8,0.8,1.0);
      else if (intensity > 0.5)
            color = vec4(0.4,0.4,0.8,1.0);
      else if (intensity > 0.25)
            color = vec4(0.2,0.2,0.4,1.0);
      else
            color = vec4(0.1,0.1,0.1,1.0);
      return(color);
}
```

Detaily jazyka

Jazyk GLSL je velmi podobný jazyku C/C++, avšak má mírné odlišnosti. Navíc GLSL obsahuje následující vlastnosti/omezení:
GLSL je 100% type-safe. Nelze tedy přiřadit integer hodnotu do float proměnné bez přetypování.

```
float my_float = 1;          // Nefunguje! 1 je integer!
float my_new_float = 1.0;    // Funguje!
```

Přetypování musí být provedeno v konstruktoru. Například následující kus kódu:

```
vec2 my_vec;
ivec2 my_int_vec;
my_vec2 = (vec2)my_int_vec;      // Nefunguje, protože není použit
                                    kosntruktor!
my_vec2 = vec2(my_int_vec);      // Funguje!
```

Vektory a matice mohou být naplněny daty jen pomocí konstruktoru:

```
vec3 my_vec = vec3(1.0, 1.0, 1.0);
mat3 my_mat = mat3(1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0);
```

Násobení vektorů je prováděno po složkách:

```
vec3 my_vec1 = vec3(5.0, 1.0, 0.0);
vec3 my_vec2 = vec3(1.0, 3.0, 4.0);
vec3 product = my_vec1 * my_vec2;   // Výsledný vektor: (5.0, 3.0,
                                       0.0)
```

Násobení vektoru a matice probíhá následovně:

*Matice * vektor* – vektor je brán jako sloupcový

*Vektor * matice* – vektor je brán jako řádkový

Vrchol je normálně transformován následujícím způsobem:

`gl_Position = myModelViewProjectionMatrix * myVertex`

Výsledkem je jednoduchá transformace pozice vrcholu maticí model-view-projection.

Můžeme (a měli bychom) využívat i řadu vestavěných funkcí, například:

dot	jednoduchý skalární součin
cross	jednoduchý vektorový součin
texture	používána pro sampling textury
normalize	normalizace vektoru
clamp	clamping vektoru na minimum a maximum